LA-UR-21-27686
Approved for public release; distribution is unlimited.

Title: Secure System Composition and Type Checking using Cryptographic Proofs

Author(s): Barrack, Daniel Abraham

Intended for: Student Symposium

Issued: 2021-08-03

Los Alamos
NATIONAL LABORATORY

Secure System Composition and Type Checking using Cryptographic Proofs

Dani Barrack
A-4: Advanced Research in Cyber Systems
Email: `dbarrack@lanl.gov`

Collaborator: Zachary DeStefano
Mentor: Michael J. Dixon
Co-Mentor: Boris Gelfand

August 3rd, 2021

ISTI Information Science & Technology Institute

Portland State UNIVERSITY

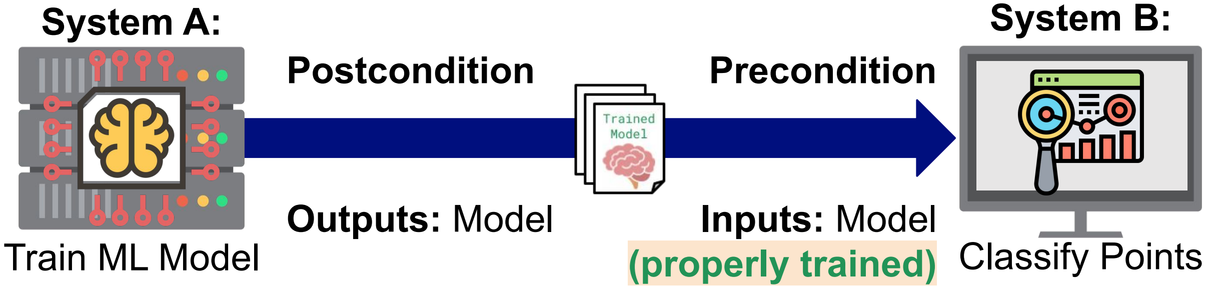
Challenge: Formally Verifying System Composition

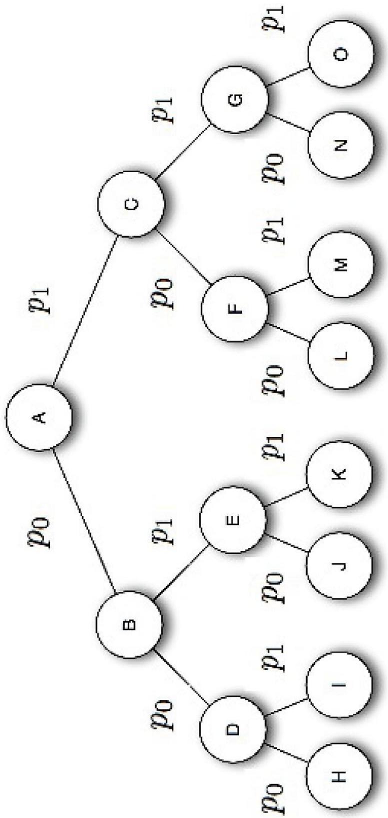
We can use formal methods to verify that systems compose correctly without the possibility of incorrect behavior.

This means exhaustively checking that System A's postconditions agree with System B's preconditions. If so, it is safe to compose.

Normal Setting: Every computational path must be accounted for and checked. Verification cost (time) is **multiplicative** across systems.

$$\text{Cost} = |S_1| \times |S_2| \times \ \dots \ \times |S_n|$$





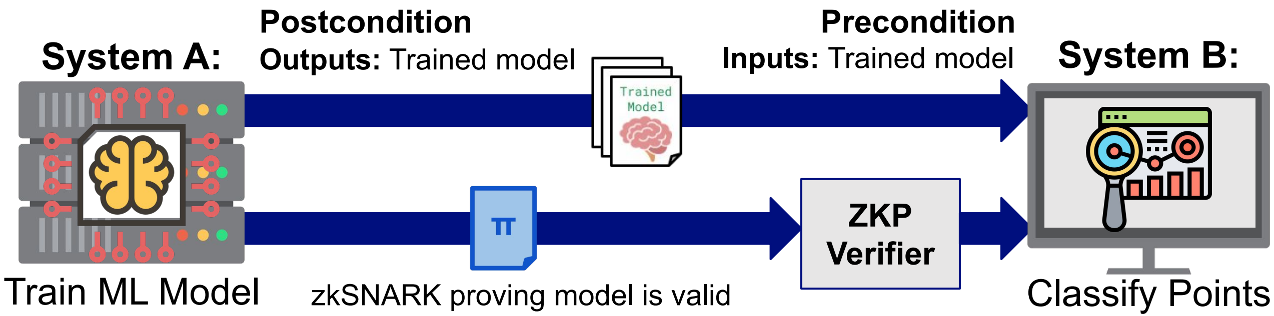
Solution: Assuring Safe Composition via zkSNARKs

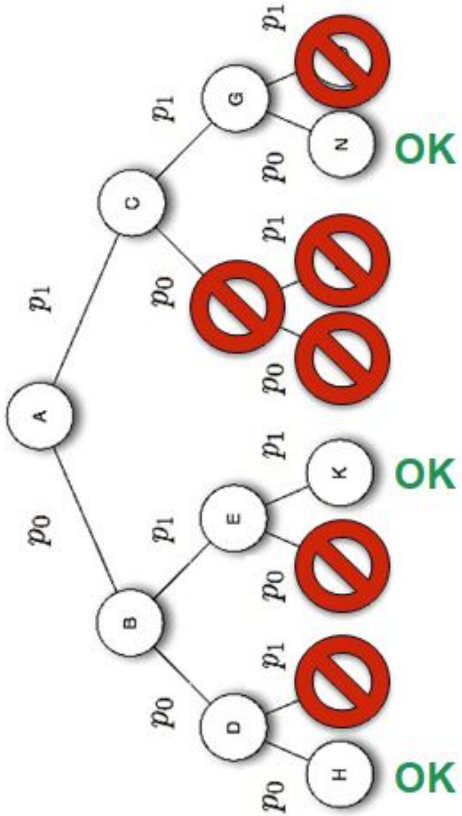
Zero-knowledge proofs (ZKPs) can be used to provide type checking guarantees of input/output properties without exposing secrets.

Verification can be done modularly so that the cost is **additive**.

$$\text{Cost} = |S_1| + |S_2| + \ldots + |S_n|$$

Bad proofs and inputs can still exist, but now are cryptographically (exponentially) hard to find and exploit.





Preconditions and Postconditions with Types

```
Type MLModel =
    (w : Weights, error(w) < 0.05, log : AuditLog, execute(log) == w)
trainModel : (x : [Input]) -> MLModel


classifyPoint : (y : Input, model : MLModel) -> Class
```

We can generate proofs (or an audit log) of desired properties (e.g. functional correctness) and include them with input to other functions.

This allows us to use a dependent type to assure that only models that were actually trained on actual data and are within a certain error threshold can be used by a classifier.

The audit log and its proof would be **huge**. Instead, we can use a super small zkSNARK to prove this dependent type and pass it along instead. We only need to handle the case where the check fails.

Preconditions and Postconditions with Types

Type MLModel =

zkp : ZKP

(w : Weights, ~~error(w) < 0.05~~, ~~log : AuditLog, execute(log) == w~~)

trainModel : (x : [Input]) -> MLModel

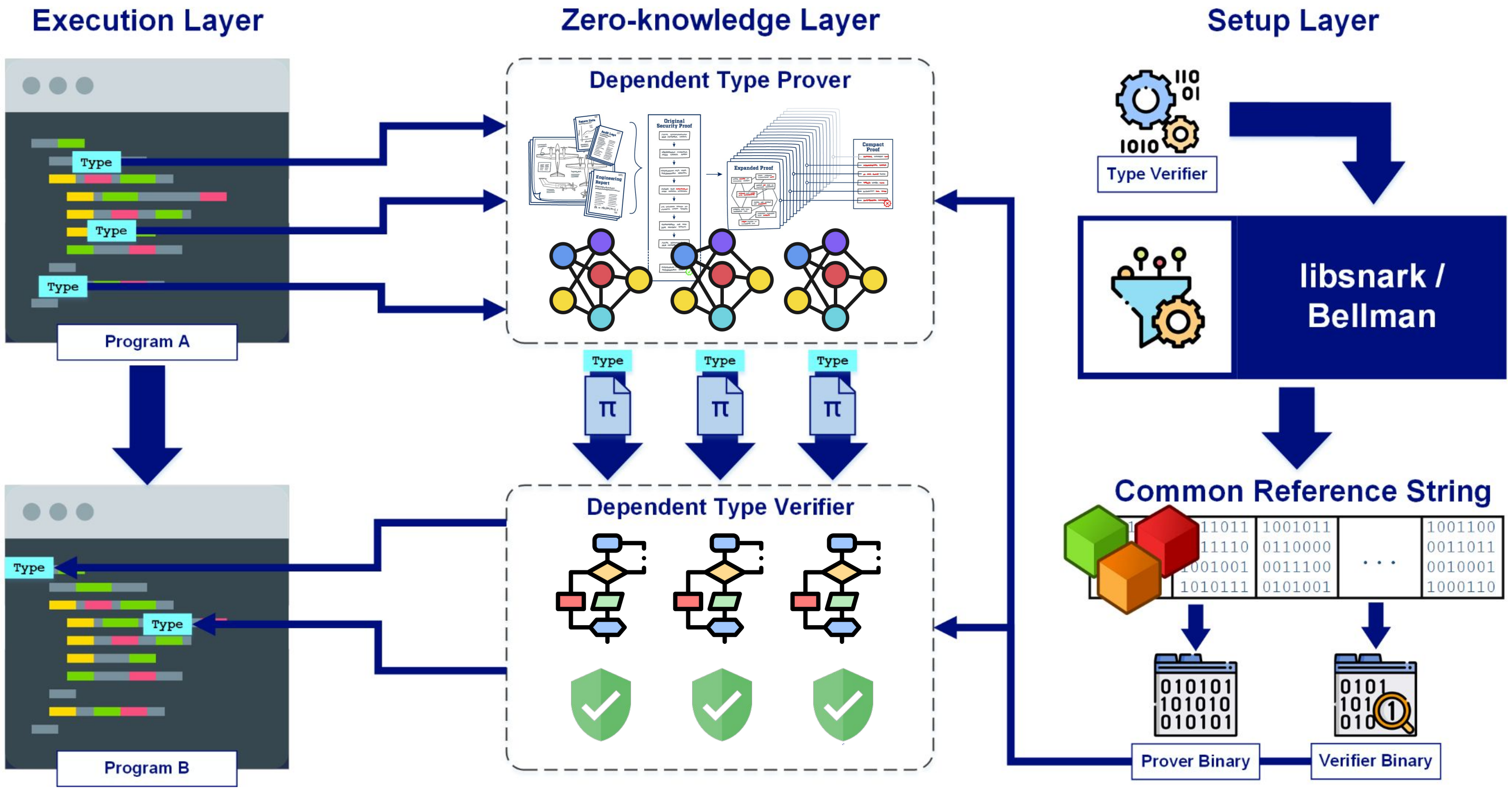
classifyPoint : (y : Input, model : MLModel, verif : ZKPVerifier) -> IO Class

We can generate proofs (or an audit log) of desired properties (e.g. functional correctness) and include them with input to other functions.

This allows us to use a dependent type to assure that only models that were actually trained on actual data and are within a certain error threshold can be used by a classifier.

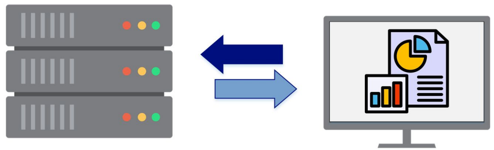
The audit log and its proof would be **huge**. Instead, we can use a super small zkSNARK to prove this dependent type and pass it along instead. We only need to handle the case where the check fails.

Dependent Type Replacement by ZKPs

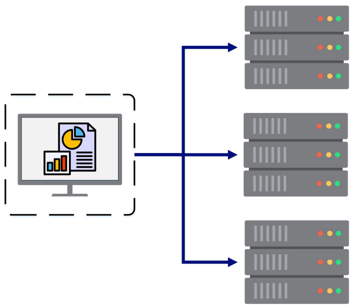


Benefits & Capabilities

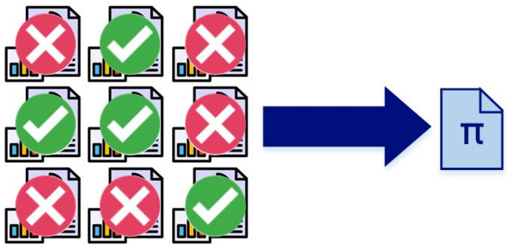
Using zero-knowledge proofs, we can combine cyber systems while preventing certain incorrect and malicious behaviors relating to mismatched outputs and input constraints.



ZKPs enforce system compatibility without the expense of manually proving correctness



Portable proofs artificially extends our trusted computing base beyond just our own system



ZKPs give fine-grained control over which bits of information to keep secret and which to prove

Implementation Summary

We developed a *library* of zkSNARK *gadgets* and *types* in `C++` using `Libsnark`

We developed a *custom compiler* in `Haskell` to apply *functional programming techniques* to zkSNARK development

We produced a demo *dependently-typed* zkSNARK application for RSA *encryption* and *verification*

Functional Gadget Library

- **RSA Components**
- **Large Integer Math**
- **Primitive Operations**
- **Map, ZipWith, Fold, ...**
- **Libsnark**

Prototype Compiler

- **Custom Domain Specific Language**
- **DSL Flattener**
- **Type Extractor**
- **Circuit Generator**
- **Haskell Generator**
- **C++ Gadget Generator**

Type Checking Demo

- **Zero-Knowledge RSA Encryption Application**
- **Zero-Knowledge RSA Verifier and Multiplier Application**
- **Application Communication Utility Scripts**

Demo:
Verifying an RSA Encryption Pipeline

Background: RSA Cryptography

Encryption: $$\mathsf{Enc}_{k_{pub}}(msg) = msg^{k_{pub}}\ mod\ N = c$$

Decryption: $$\mathsf{Dec}_{k_{priv}}(c) = c^{k_{priv}}\ mod\ N = msg$$

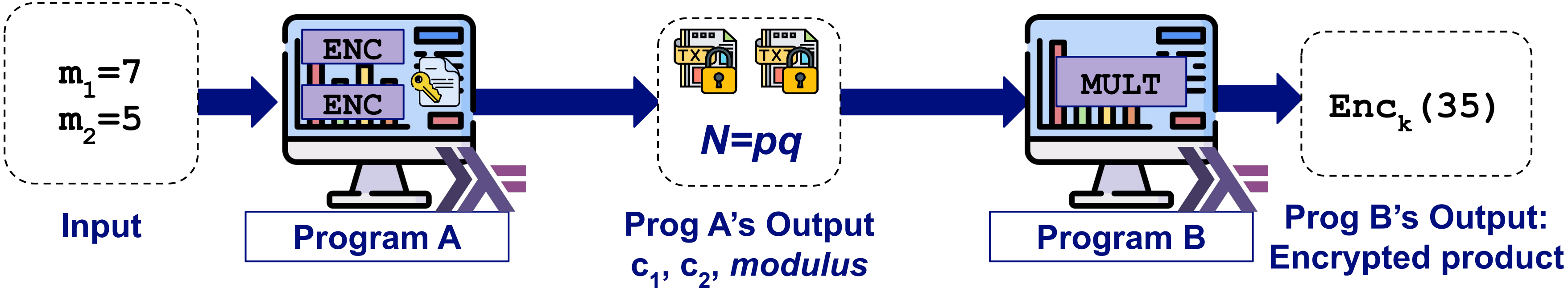
RSA is multiplicatively (×) homomorphic, meaning that if we encrypt two messages with the same key and modulus, the multiplication of those two ciphertexts equals the encryption of the multiplication of the plaintexts

$$\begin{aligned} c_1 * c_2 &\equiv msg_1^{k_{pub}} * msg_2^{k_{pub}} \mod N \\ &\equiv (msg_1 * msg_2)^{k_{pub}} \mod N \\ &\equiv \mathsf{Enc}_{k_{pub}}(msg_1 * msg_2) \end{aligned}$$

Demo: RSA Encryption Pipeline

Sample Pipeline

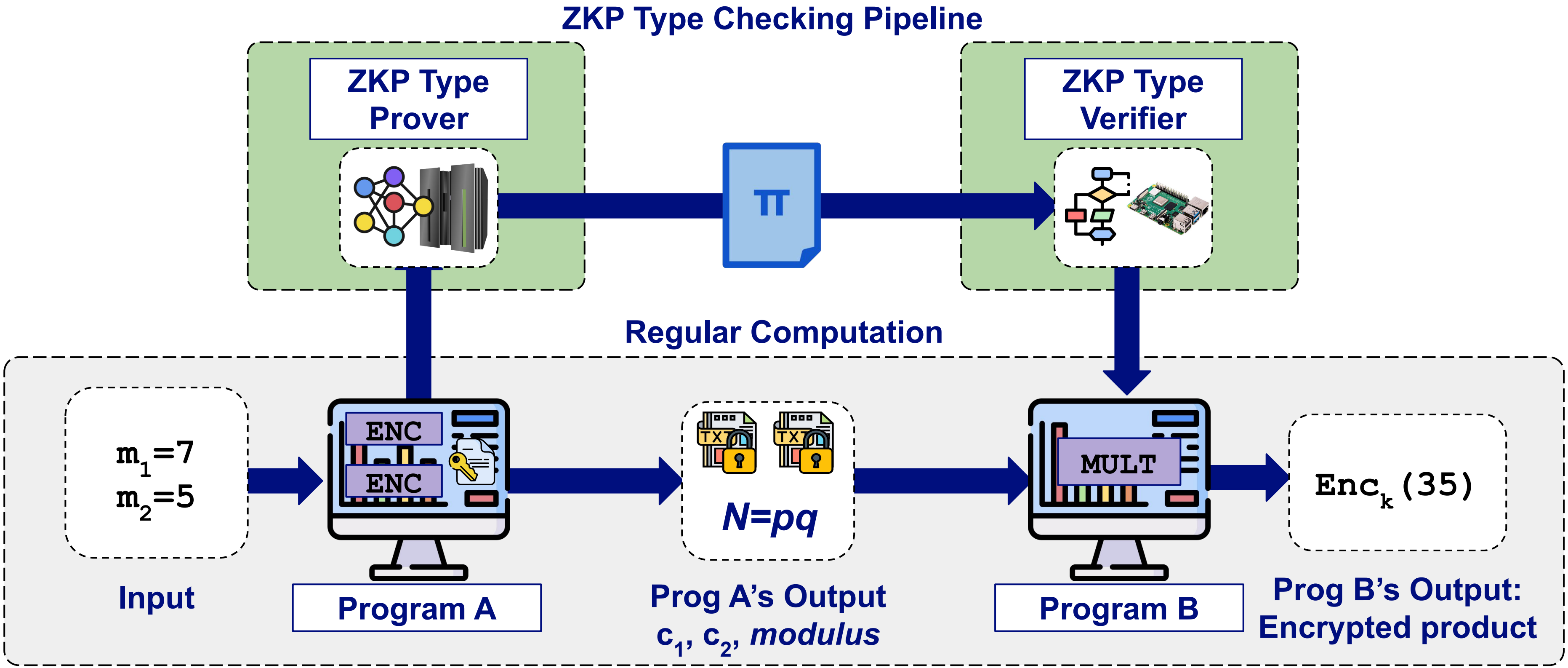
1. Program A encrypts two secret messages using RSA
2. Program B receives encrypted messages and multiplies them



Challenge

If we implement **A** and **B** in Haskell, program **B** can't guarantee it is multiplying valid RSA ciphertexts. **B** could end up yielding garbage and would be an **error** a type checker could catch **IF** it could see everything 1) only discoverable at runtime and 2) under the covers of encryption.

Demo: Encryption Pipeline with Type Checking ZKPs



Demo: Proving Type Checks with ZKPs

Haskell's type checker can't verify the encrypted variable's type until program runtime. Instead, we instruct it to know to ask for a ZKP of its type later.

Example. Type for a valid pair of RSA ciphertexts

```
type ValidRSAPair =
EncRSA(key, modulus, message1) == cipher1
and
EncRSA(key, modulus, message2) == cipher2
```

We can encode this proof as the type ValidRSAPair above, and generate a zkSNARK that proves type compliance using our compiler toolchain.

We can use Type-Level Haskell to generate redacted and un-redacted types, so type information is not lost between function calls, but sensitive information is not present.

A function to Illustrate Homomorphic Property

```
encryptMessagePair ::
Length
-> Message
-> Message
-> PublicKey
-> Modulus
-> IO (Redacted RSAPair)
```

```
multiplyPair ::
IO (Redacted RSAPair)
-> PrivateKey
-> Integer
-> Message
```

This function encrypts two messages with the same key and modulus, and returns them along with the bit width.

The decrypt function relies on the fact that the two supplied ciphertexts are encrypted with the same key and modulus.

Demo: Unredacted Pair Multiplier

```
multiplyPair ::
(Length, Message, Message, Key, Mod, CipherText, CipherText)
-> PrivateKey
-> Integer
-> IO Message
multiplyPair r@(bits,m1,m2,pubKey,modulus,c1,c2) = do
  verifyZKP r
  prod <- (c1 * c2) `mod` modulus
  return prod
```

A non- redacted multiplication function input reveals sensitive information

Demo: Unredacted Pair Multiplier

```
multiplyPair ::
(Length, Message, Message, Key, Mod, CipherText, CipherText)
-> PrivateKey
-> Integer
-> IO Message
multiplyPair r@(bits,[redacted],[redacted],[redacted],[redacted],c1,c2) = do
  verifyZKP r
  prod <- (c1 * c2) `mod` modulus
  return prod
```

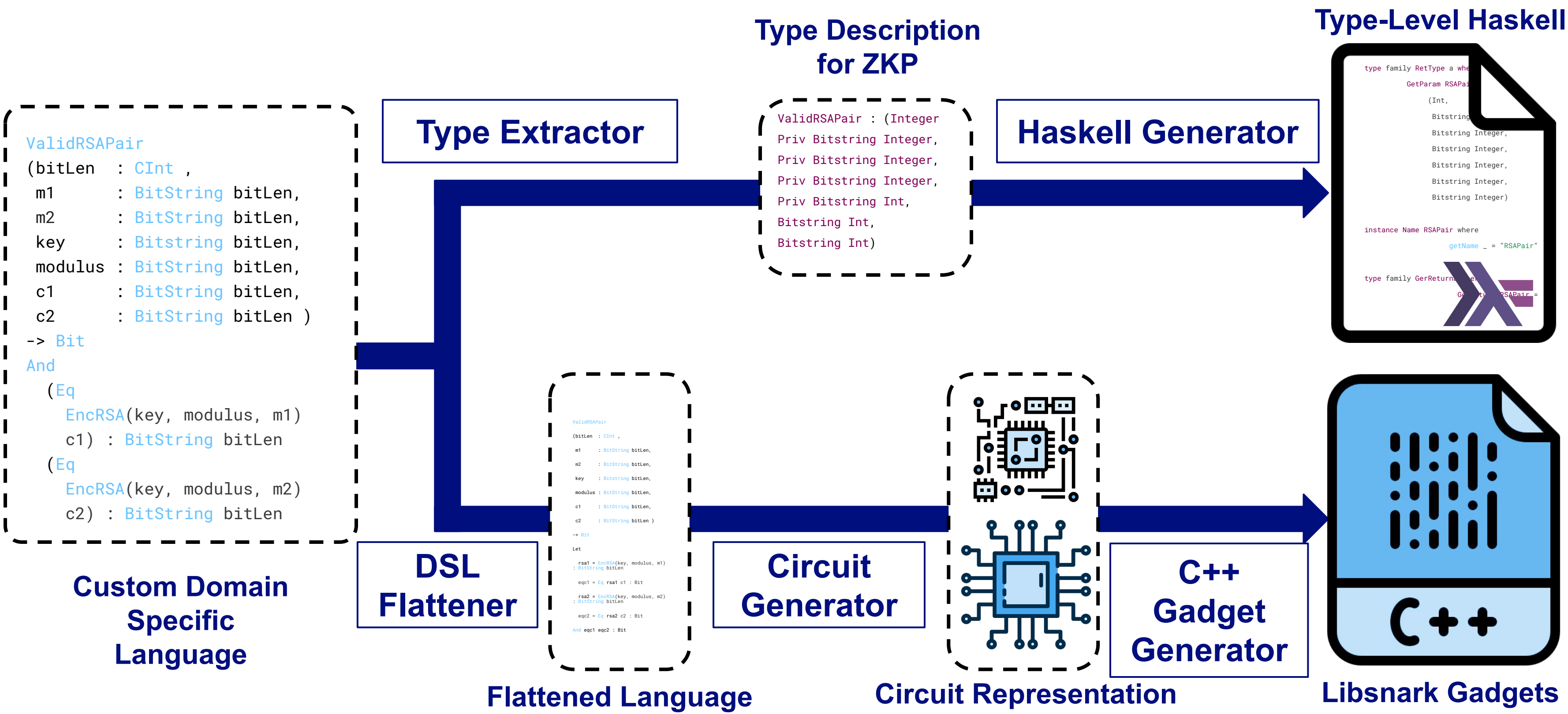
The redacted information is simply not available when passed as an input.

Demo: Redacted Pair Multiplier

```
multiplyPair ::
Redacted RSAPair
-> PrivateKey
-> Integer
-> IO Message
multiplyPair r@(bits,_,_,_,_,c1,c2) = do
  verifyZKP r
  prod <- (c1 * c2) `mod` modulus
  return prod
```

The redacted information is simply not available when passed as an input.

Full Compiler Pipeline



Conclusion

By using zkSNARKs to prove that values have specific **dependent types**, it is possible to provably assure compatibility and correctness without revealing sensitive information and extend our trusted computing base well beyond our own system.

The approach we developed expands the scope of what non-interactive zero-knowledge proofs can capture to include properties about both the execution and correctness of programs

Future Work

I. Increase the extent of Haskell language integration to **enforce verification on a programming language level** rather than trusting programmers to run the verifier binaries externally.

II. Leverage approach to work with several ongoing efforts at LANL to help **verify mission-relevant cyber systems** that utilize sensitive information

III. Build more advanced compiler **automation** to automatically integrate type-level haskell and compile libsnark programs to allow faster development times.

IV. Build **optimization** steps to reduce number of gates into the compiler, and optimize existing gadgets.

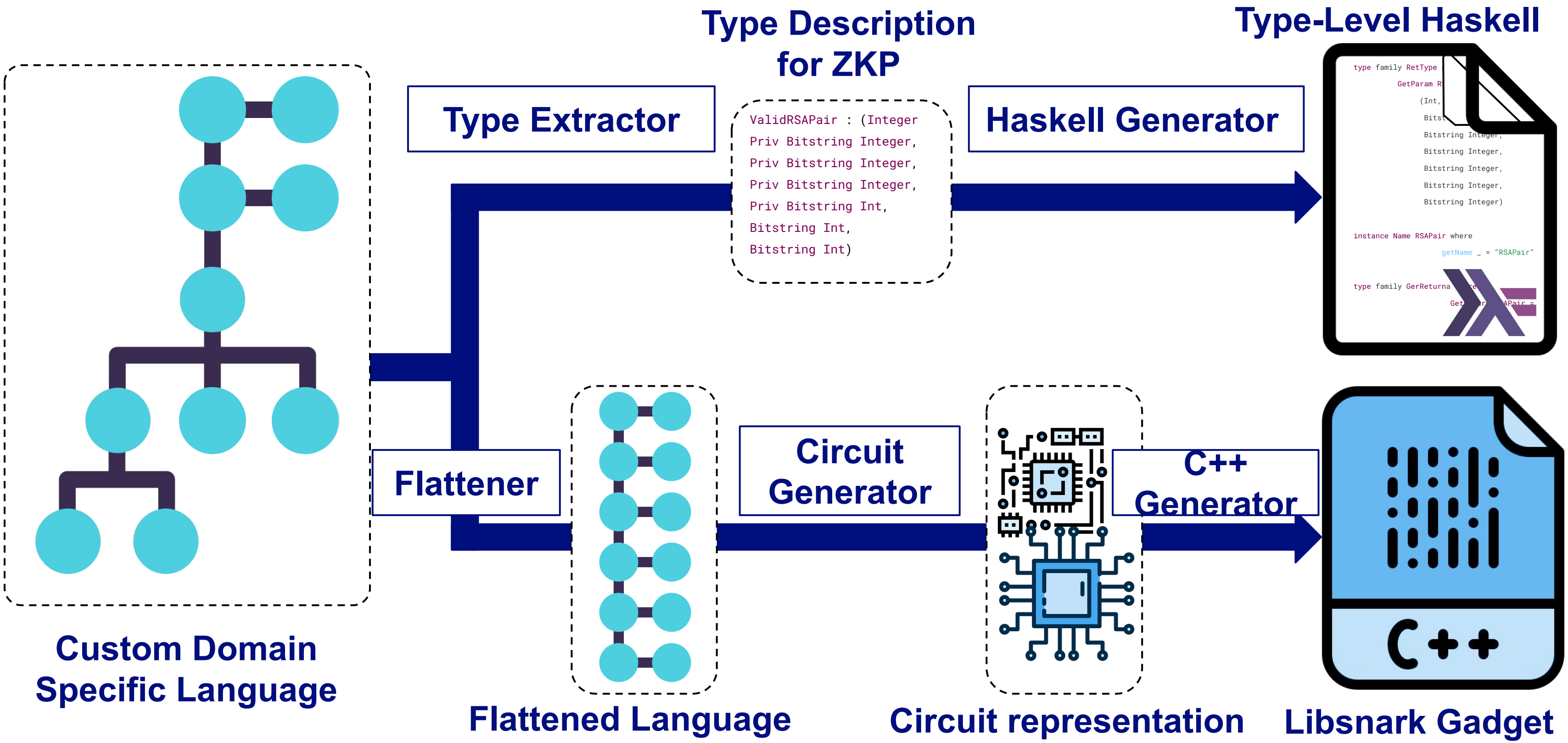
V. Increase the expressivity of the language to include ZKPs for **uncertainty measures** and **machine learning model properties** developed by fellow LANL student, Zachary DeStefano (A-4).

Questions?

dbarrack@lanl.gov

Backup

Full Compiler Pipeline



Demo: Flattened Language Example

```
ValidRSAPair
(bitLen   : CInt ,
 m1       : BitString bitLen,
 m2       : BitString bitLen,
 key      : Bitstring bitLen,
 modulus  : BitString bitLen,
 c1       : BitString bitLen,
 c2       : BitString bitLen )
-> Bit
Let
   rsa1 = EncRSA(key, modulus, m1) : BitString bitLen
   eqc1 = Eq rsa1 c1 : Bit
   rsa2 = EncRSA(key, modulus, m2) : BitString bitLen
   eqc2 = Eq rsa2 c2 : Bit
And eqc1 eqc2 : Bit
```

Demo: Circuit Example

```
ValidRSAPair<FieldT>
(bitLen   : Int,
 m1       : pb_variable_array<FieldT>,
 m2       : pb_variable_array<FieldT>,
 key      : pb_variable_array<FieldT>,
 modulus  : pb_variable_array<FieldT>,
 c1       : pb_variable_array<FieldT>,
 c2       : pb_variable_array<FieldT> )
-> pb_variable<FieldT>
Wires:
  rsa1 : pb_variable_array<FieldT>, bitLen
  rsa2 : pb_variable_array<FieldT>, bitLen
  eqc1 : pb_variable_array<FieldT>, bitLen
  eqc2 : pb_variable_array<FieldT>, bitLen
Gates:
  EncRSA(key, modulus, m1)
  Eq rsa1 c1 : Bit
```

Demo: Custom Domain Specific Language Example

```
ValidRSAPair
(bitLen   : CInt ,
 m1       : BitString bitLen,
 m2       : BitString bitLen,
 key      : Bitstring bitLen,
 modulus  : BitString bitLen,
 c1       : BitString bitLen,
 c2       : BitString bitLen )
-> Bit
And
  (Eq EncRSA(key, modulus, m1) c1) : BitString bitLen
  (Eq EncRSA(key, modulus, m2) c2) : BitString bitLen
```

Shared Key and Modulus as a Type

```
type RSAPair =
RSA(key, modulus, message1) == cipher1
and
RSA(key, modulus, message2) == cipher2
```

With traditional proofs we have a choice, either supply the key and the modulus that encrypted them so the receiver side can manually verify this property, or trust that the input was prepared correctly and risk incorrect behavior.

We can encode this proof as the type “RSAPair” above, and generate a zkSNARK to capture this property.

Multiplicatively Homomorphic Property of RSA

If we encrypt two messages with the same key and modulus, the multiplication of those two ciphertexts equals the encryption of the multiplication of the plaintexts

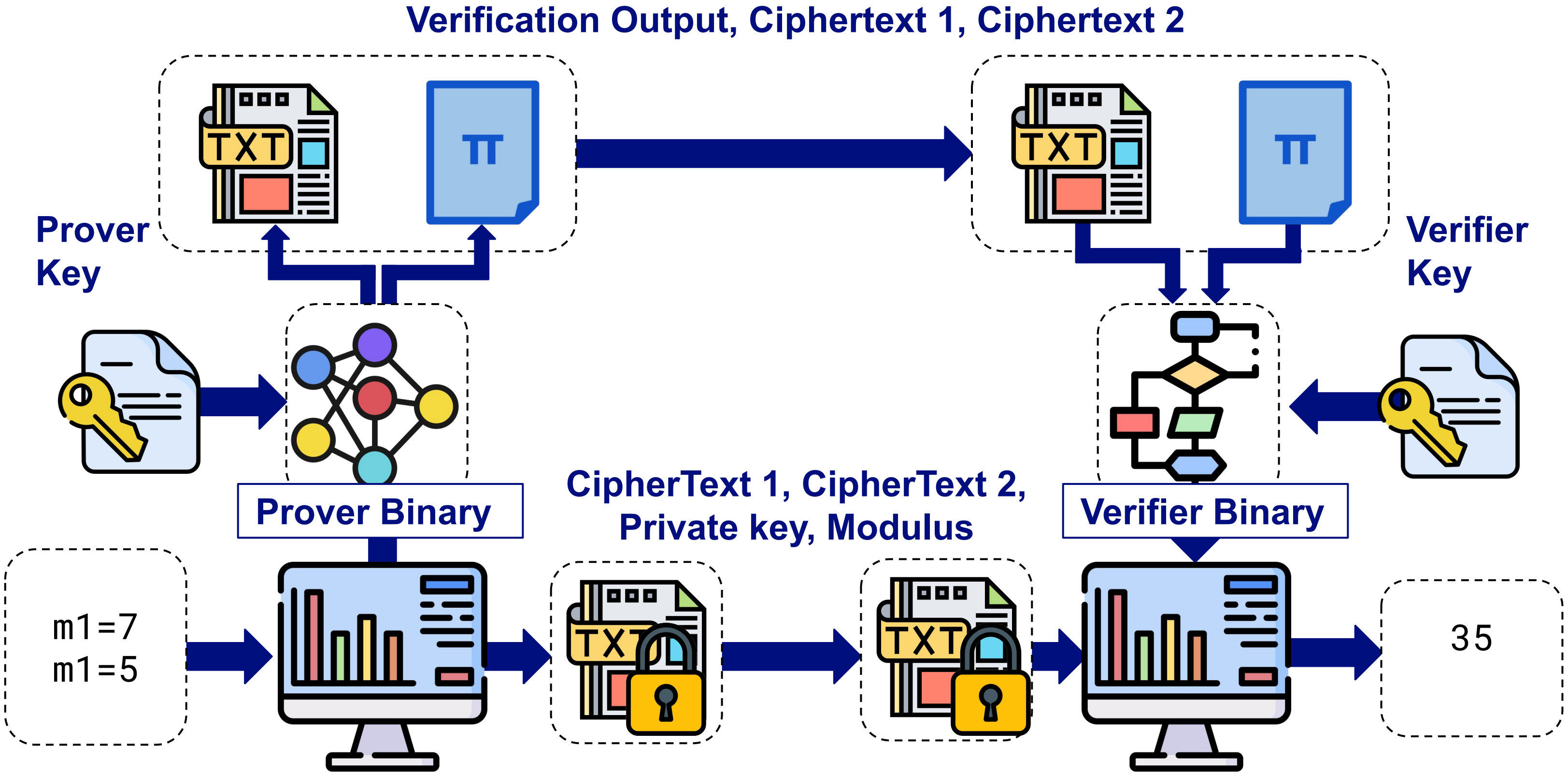
$$msg_1^{k_{pub}} \bmod N = c_1$$

$$msg_2^{k_{pub}} \bmod N = c_2$$

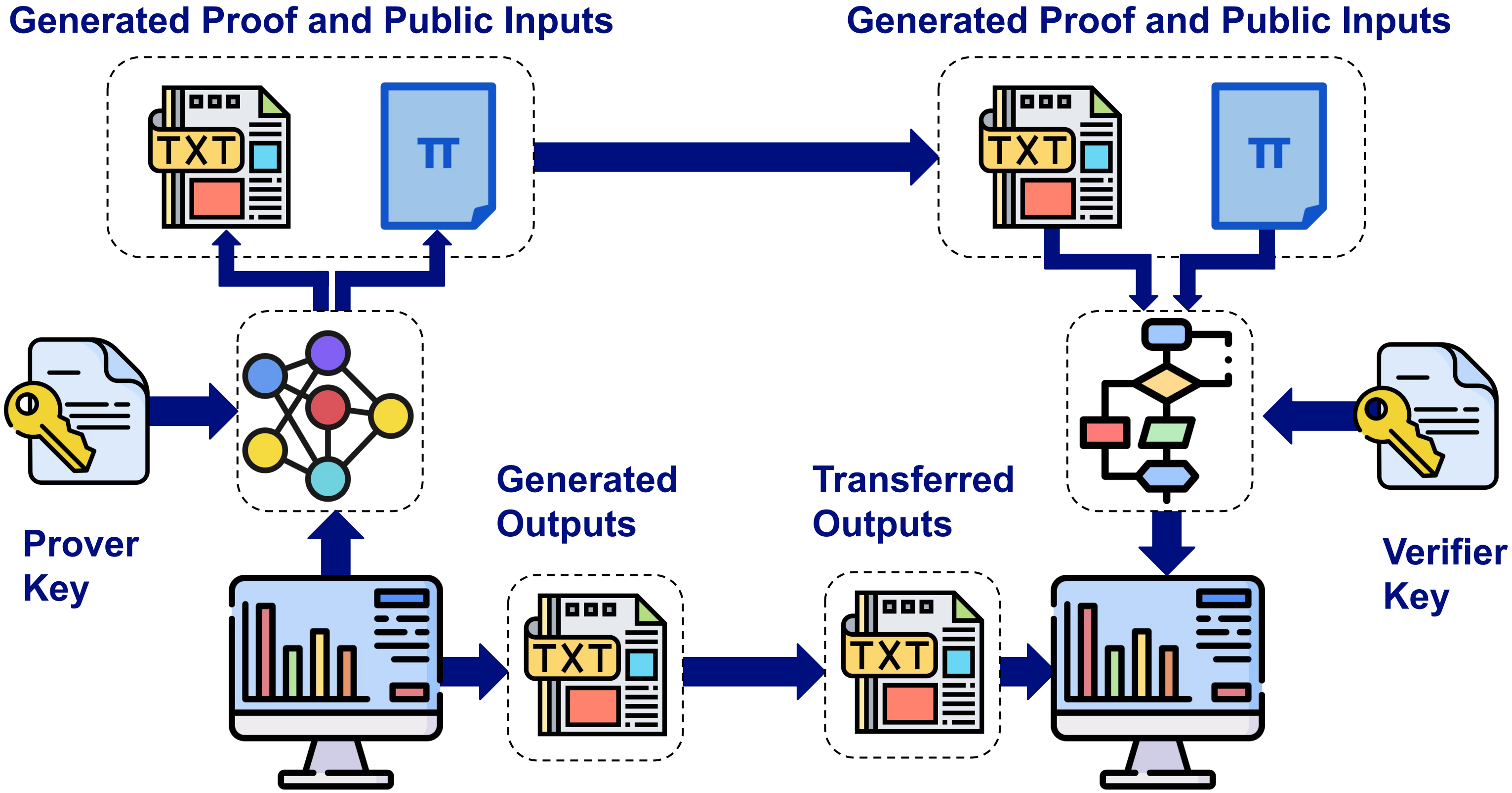
$$\mathsf{Encrypt}(key, N, msg_1) * \mathsf{Encrypt}(key, N, msg_2) \bmod N$$

$$\equiv \mathsf{Encrypt}(key, N, msg_1 * msg_2)$$

Distributed Verification



Distributed Verification

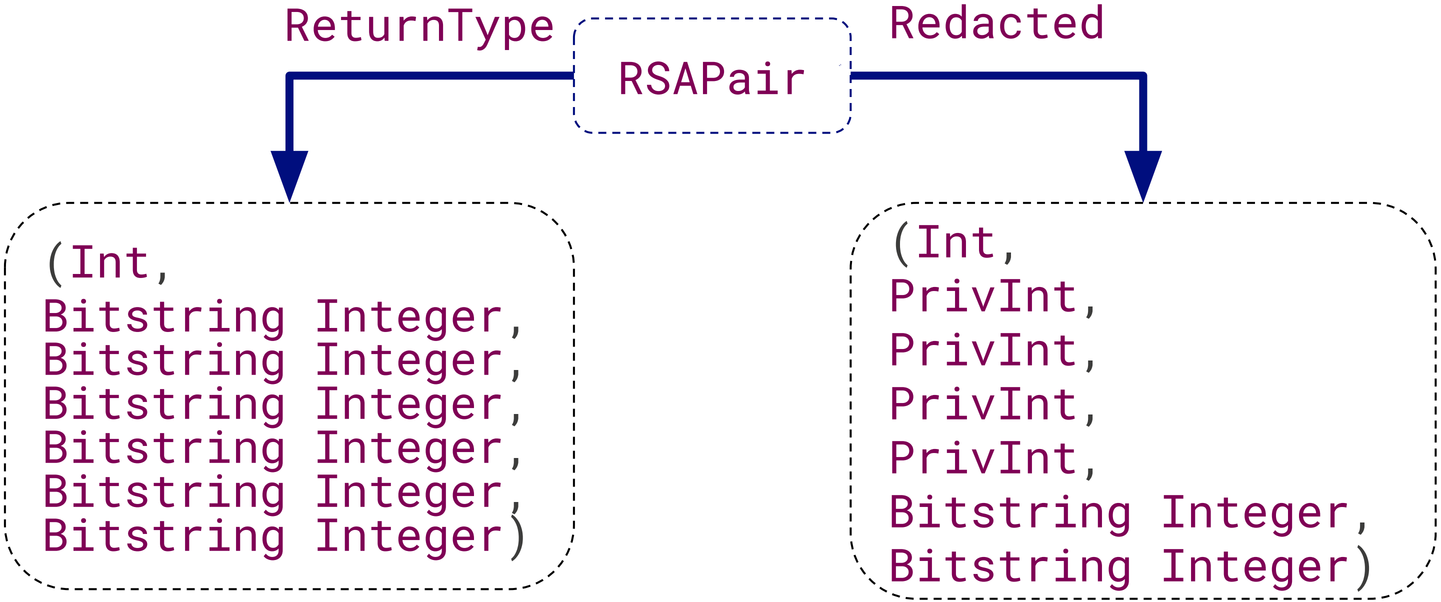


Project Roadmap

1. Prototype and test program interoperability
   – Manually implement skeleton code in place of zero-knowledge proofs to interact with example program
2. Implement constraint related ZKP gadgets
   – Constraints capture type information that is immediately useful to the test program
3. Develop and set up prototype demonstrations
   – Replace skeleton code with handcrafted ZKP gadgets
   – Develop prototype compiler to read type annotations from file and generate constraints
4. Benchmark and Evaluate

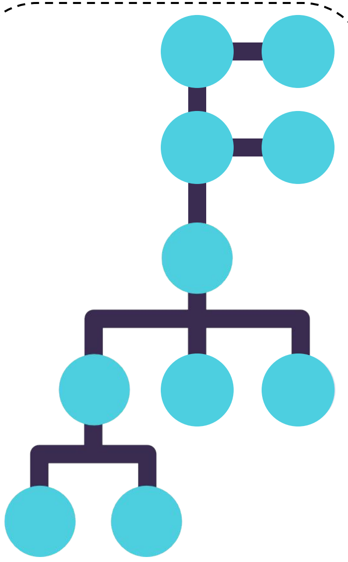
Prototype ZKP Compiler

Type-Level Programming Gives Type Safety

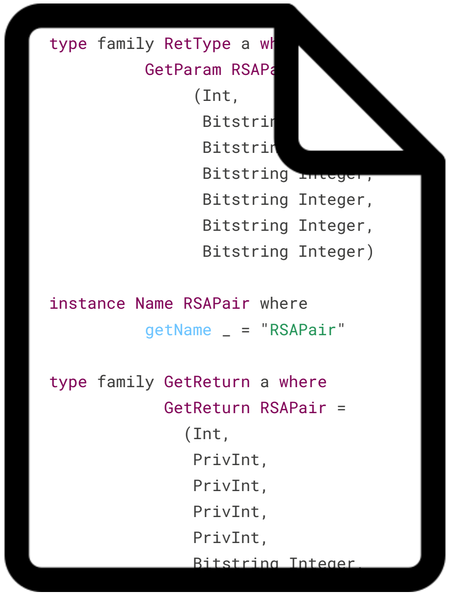


We can use type level programming to generate input and output types for functions from a central type.

Type-level Haskell Generation Step

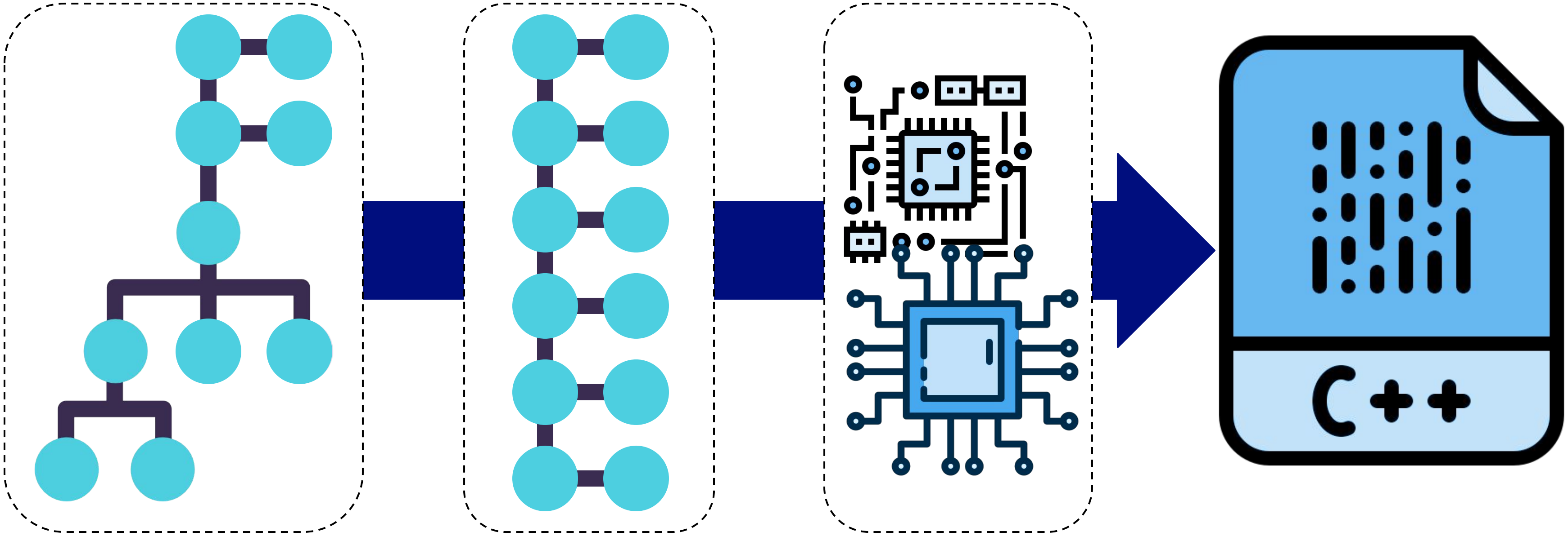


```
RSAPair : (Int
Priv Bitstring Integer,
Priv Bitstring Integer,
Priv Bitstring Integer,
Priv Bitstring Int,
Bitstring Int,
Bitstring Int)
```



Expression **zkSNARK Type** **Type-Level Haskell**

Compiler intermediate representations



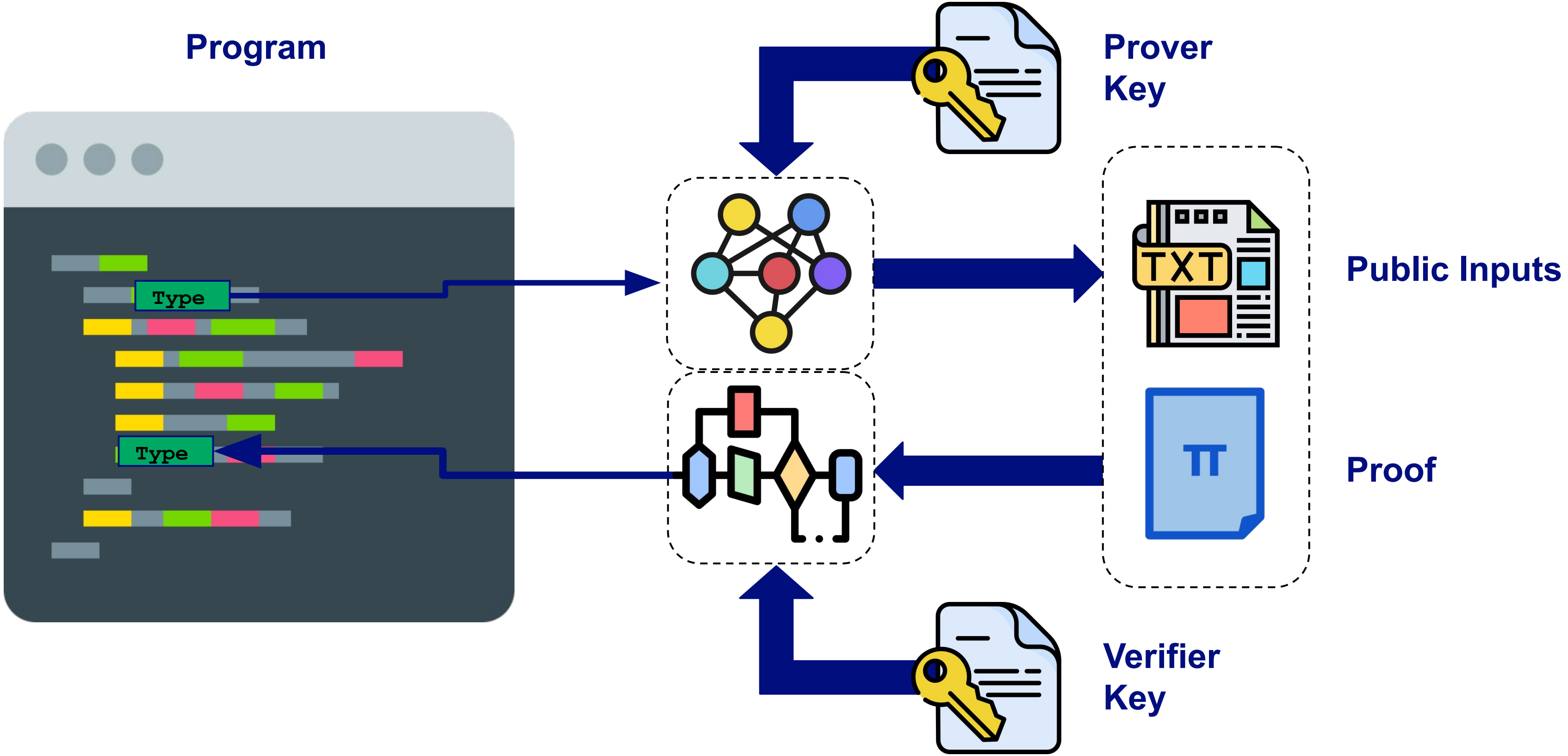
Language

Flattened Language

Circuit representation

Libsnark Gadget

Verification in a program



A function to Illustrate Homomorphic Property

```
encryptMessagePair ::
Length
-> Message
-> Message
-> PublicKey
-> Modulus
-> (Length, CipherText, CipherText)
encryptMessagePair bits m1 m2 key modulus =
   c1 <- encrypt m1 key modulus
   c2 <- encrypt m2 key modulus
   return (bits, c1, c2)
```

This function encrypts two messages with the same can modulus, and returns them along with the bit width.

A function to Illustrate Homomorphic Property

```
multiplyDecryptPair ::
(Length, CipherText, CipherText)
-> PrivateKey
-> Integer
-> Message
multiplyDecryptPair (bits,c1,c2) key modulus = do
  prod <- (c1 * c2) `mod` modulus
  c3   <- decrypt prod key modulus
  return c3
```

This function multiplies to ciphertext together, then decrypts it with the given key.

Its correct functioning depends on the two ciphertexts having been encrypted with the same key and modulus.

Encryption Property Verified with ZKP

```
encryptMessagePair ::
Length
-> Message
-> Message
-> PublicKey
-> Modulus
-> IO (Redacted RSAPair)
encryptMessagePair bits m1 m2 key modulus =
   c1 <- encrypt m1 key modulus
   c2 <- encrypt m2 key modulus
   prepareZKP (bits, m1,m2, key, modulus, c1, c2)
```

This prepares the ZKP, which generates the proof files and redacts the information we don't want the other function to see.

A function to Illustrate Homomorphic Property

```
multiplyDecryptPair ::
(Redacted (RSAPair))
-> PrivateKey
-> Integer
-> IO Message
multiplyDecryptPair param@(bits,_,_,_,_,c1,c2) key modulus =
do
  verifyZKP param
  prod <- (c1 * c2) `mod` modulus
  c3   <- decrypt prod key modulus
  return c3
```

We verify the ZKP before multiplying the ciphertexts. If verification fails, an error is thrown

Demo Summary and Challenges

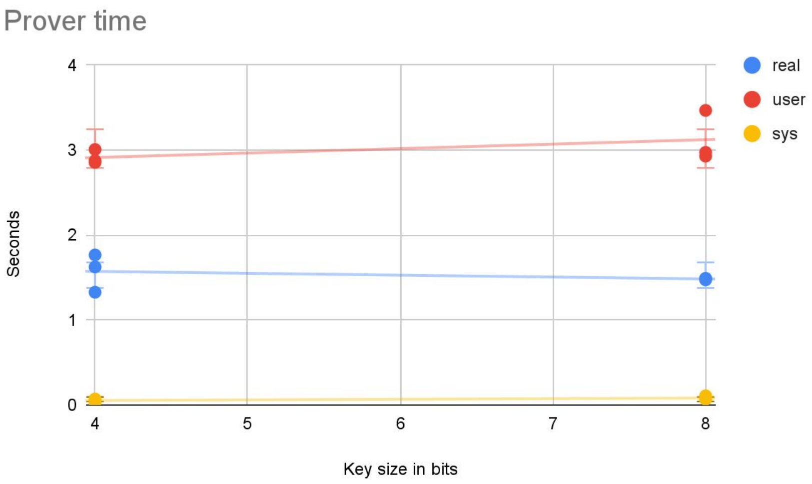
We were able to show an example of this approach using zkSNARKS to verify both functions interacting in a program, and programs interacting across a file system.

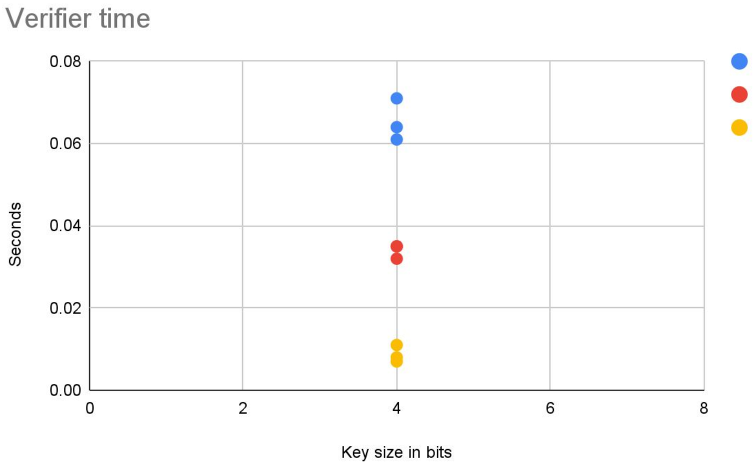
It relies on the writing of zkSNARK gadgets, which using extant libraries is extremely labor-intensive and requires knowledge of esoteric programming techniques.

In order to leverage the Haskell type system this approach requires type level Haskell programming, which is considered niche even among advanced Haskell programmers.

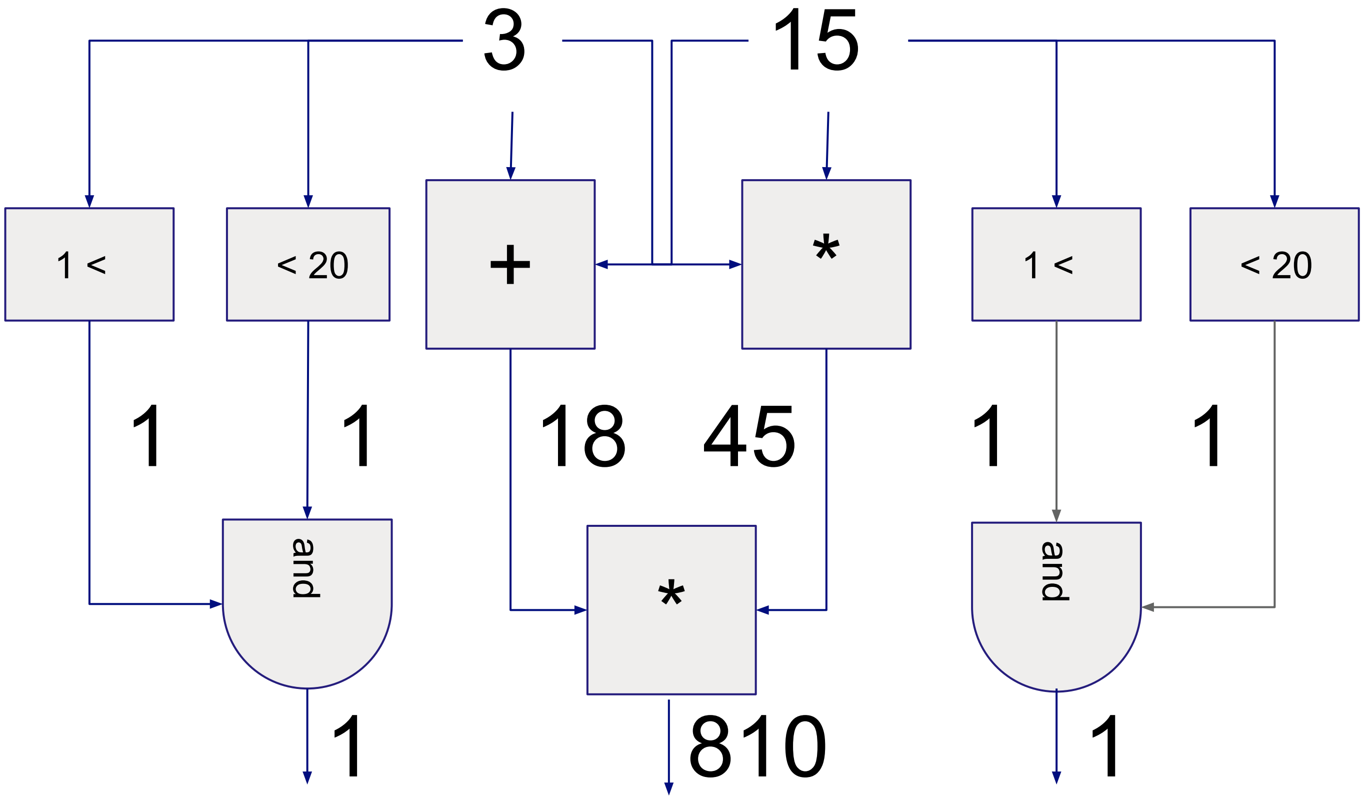
Can we mitigate these challenges?

Benchmarks for Demo

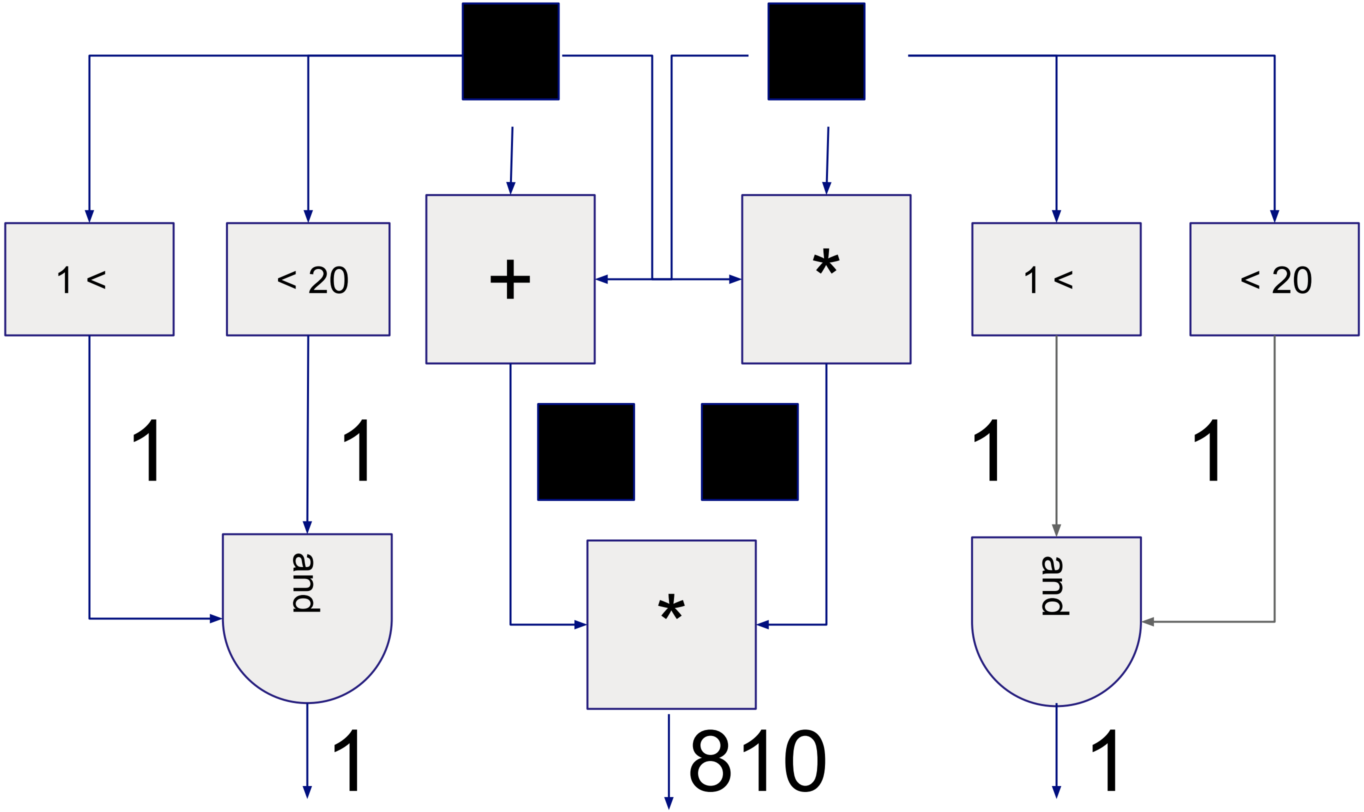




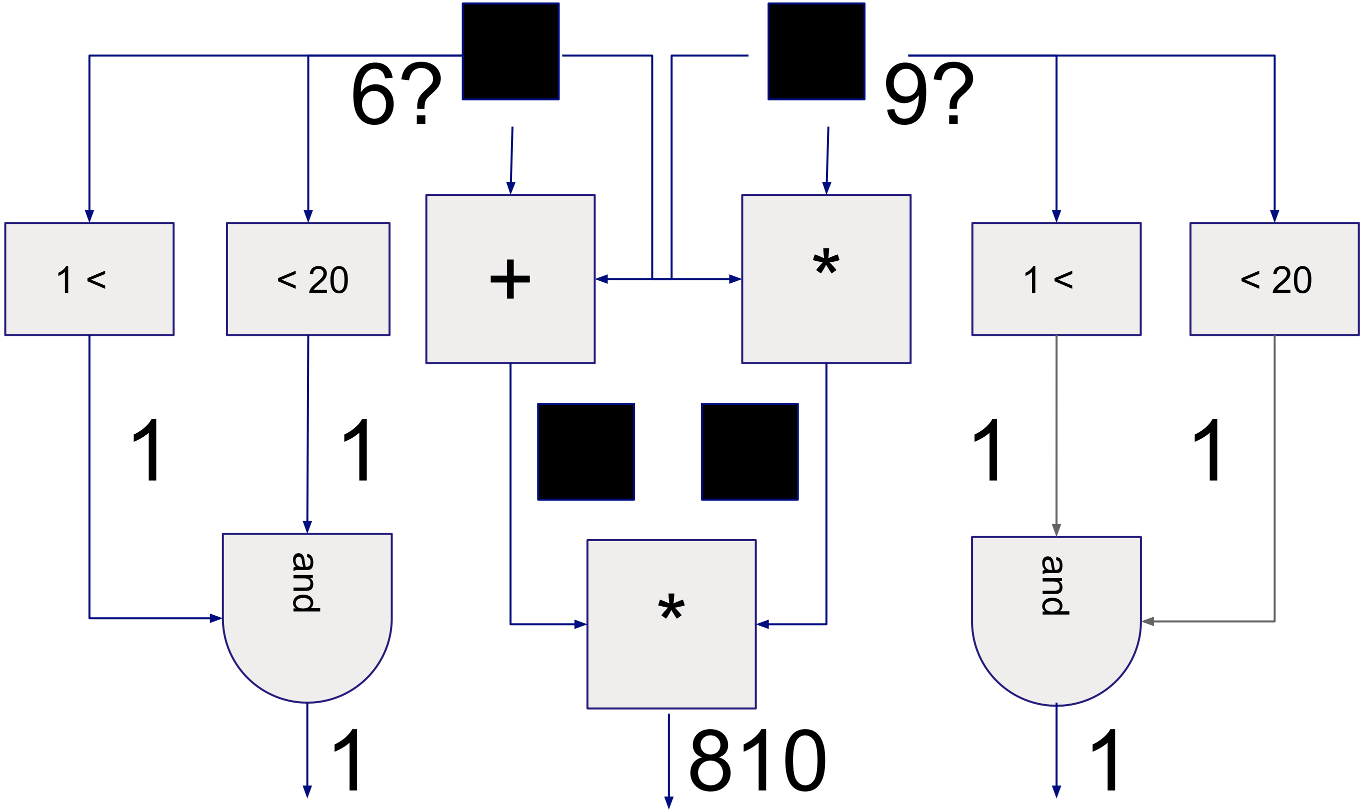
Our function as a circuit



A redacted circuit with zkSNARKs



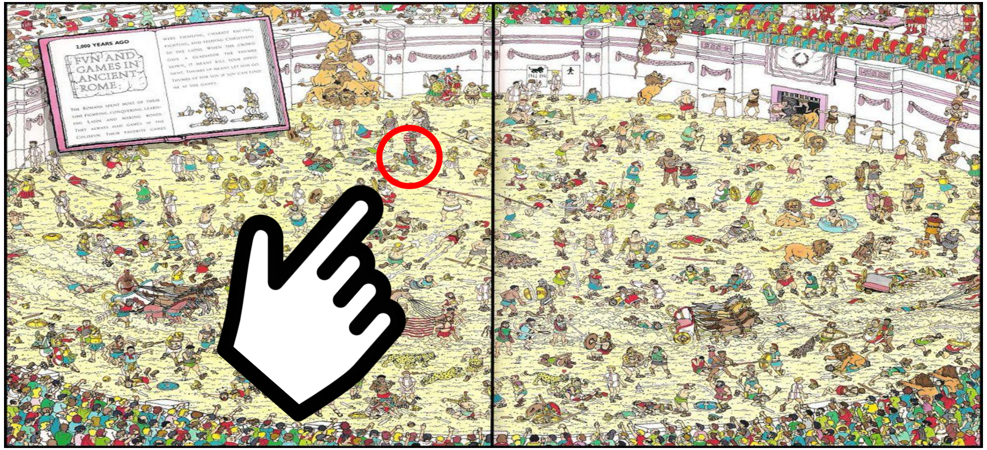
A redacted circuit with zkSNARKs



Zero-Knowledge Proof for *Where's Waldo?*

Example. You want to prove that you have beaten *Where's Waldo?*

- **Traditional Proof:** Point to Waldo to demonstrate you know where he is



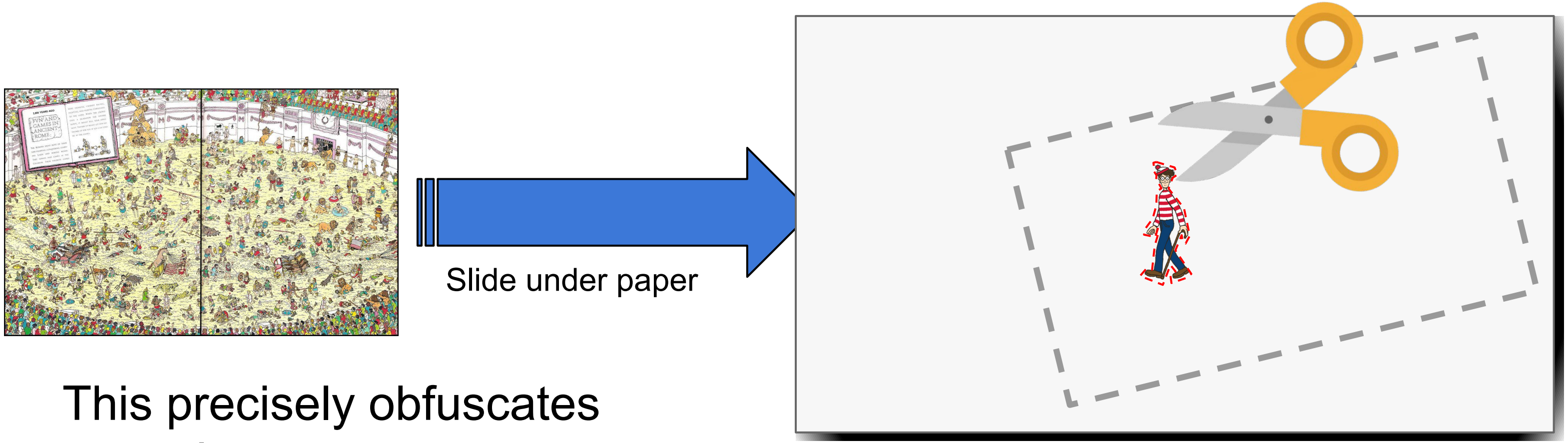
- Not zero-knowledge!

  This kind of proof leaks all information about his location, much more than simply that you have *knowledge* of the location

Zero-Knowledge Proof for *Where's Waldo?*

Zero-knowledge proof for "Where's Waldo?"

1. Cut out a Waldo shaped hole in a much larger piece of paper
2. Position the hole over Waldo's location



This precisely obfuscates Waldo's location while demonstrating knowledge of his whereabouts!

To adversaries, the book underneath could hypothetically be in any random orientation

Completeness vs. Soundness

Typical proof systems have 100% completeness and 100% soundness

Completeness: $\mathbb{P}$[true statement AND verifier accepts] = 1
"Everything true is provable"

Soundness: $\mathbb{P}$[false statement AND verifier rejects] = 1
"False statements aren't provable"

Cryptographic Proof Systems

Cryptographic proof systems have variable completeness and soundness. For non-interactive zero-knowledge proofs we care about:

(Completeness) $\mathbb{P}$[true statement AND verifier accepts] = 1
"Everything true is provable"

(Soundness) $\mathbb{P}$[false statement AND verifier rejects] = 1 - ε
"Low chance that a proof of a false statement is encountered"

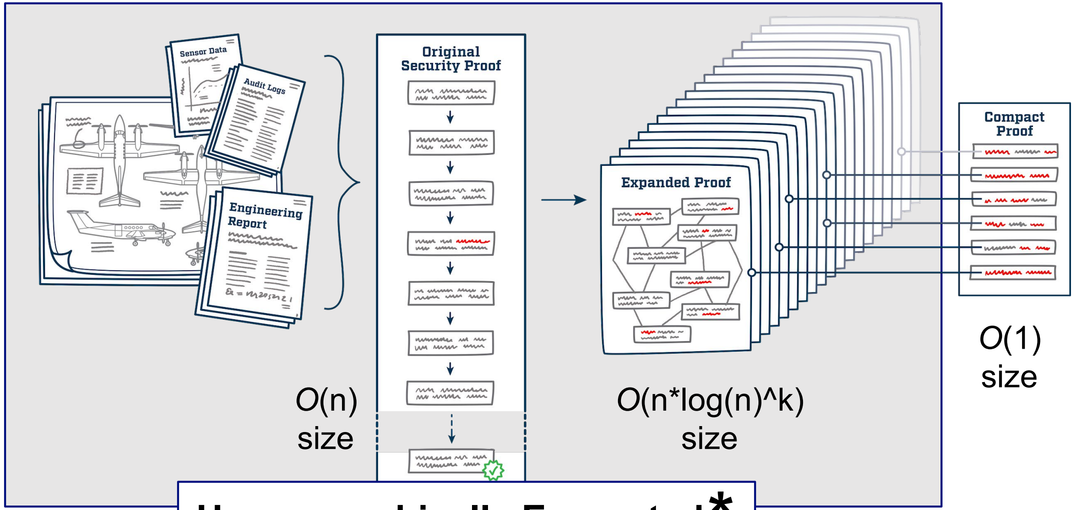
We sacrifice minimal amount of soundness (have to break crypto to produce counter-example) in order to get valuable proof properties

Zero-Knowledge Proofs and Verifiable Computation

Zero-knowledge proofs (ZKPs) allow us to prove that a claim **IS** true without revealing **WHY** it is true, even if the prover is untrusted and malicious.

zkSNARKs are special ZKPs that are ***tiny*** and ***non-interactive***

Inputs:
Logs
Schematics
Program Traces
Signals
Encryption Keys
Attestations
etc.



Homomorphically Encrypted*

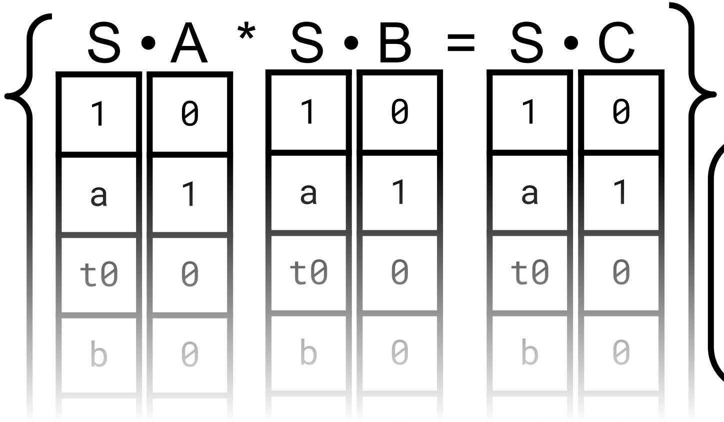
Outputs:
Results
&
ZK Proof of Computational Integrity

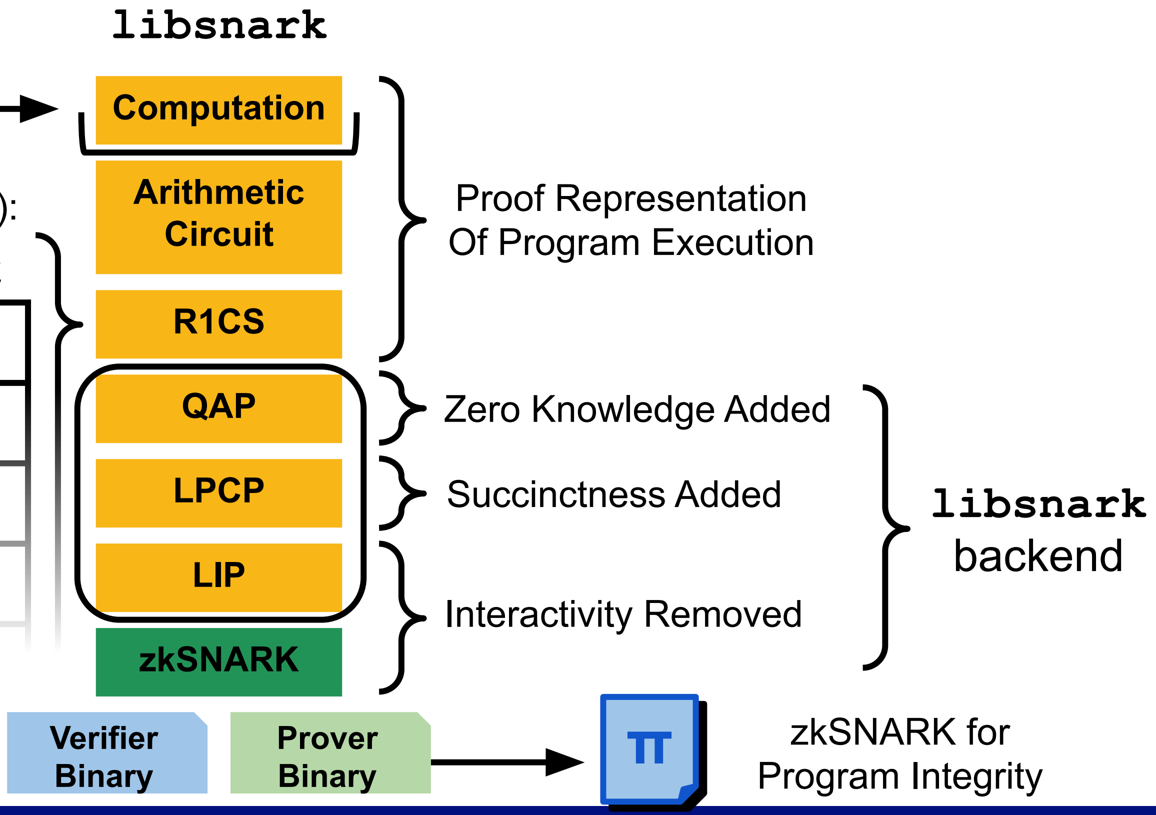
***with tweaks**

zkSNARK Construction for Program Verification [BCGTV13]

```
int myFunction(int a) {
     int b=a*a-4;
     return 3*b+a;
}
```

Rank-1 Constraint System (R1CS):





Theory Behind ZKPs (Backup)

PCPs & Hardness of Approximation

Intuition
Efficient approximation scheme for a problem implies that it an easy to create a good enough looking “fake” solution (witness)

So,

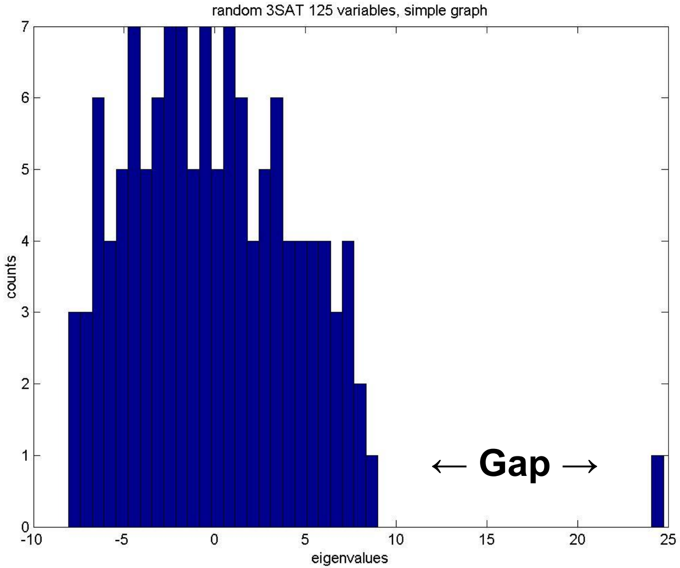
Hard to approximate

$\Leftrightarrow$

Hard to create a convincing fake witness that appears optimal

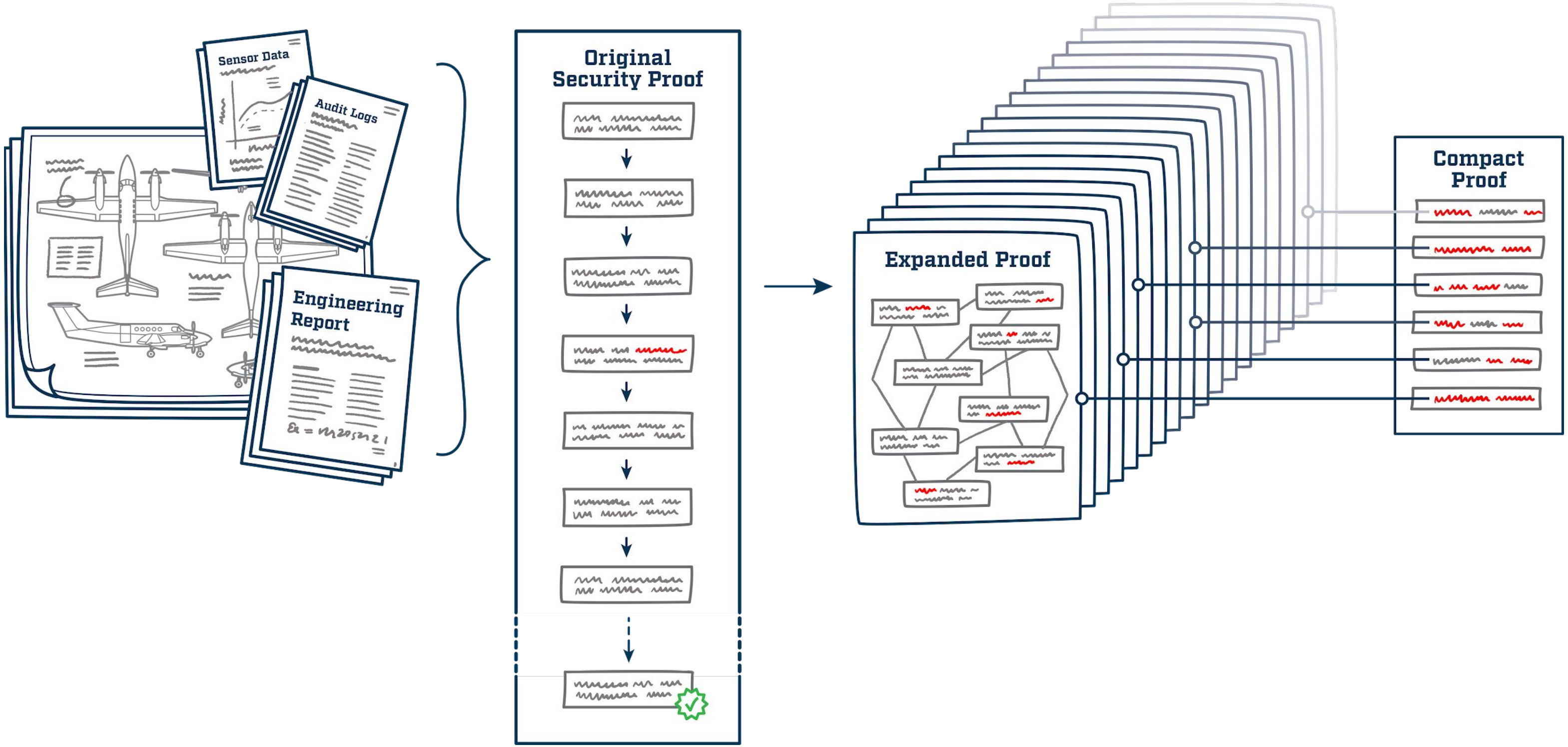
$\Leftrightarrow$

Good witnesses imply that best solutions exist



Easy to find Hard to find

NIZK Overview



Circuit Evaluation

1. Publish homomorphically encrypted building blocks for a program

$$\mathrm{CRS} = \{\mathrm{Enc}_k(\alpha x), \mathrm{Enc}_k(\alpha x^2), \ldots, \mathrm{Enc}_k(\alpha x^d)\}$$

2. Prover blindly re-assembles them to compute the desired circuit (e.g. an evaluation of the PCP circuit) and adding random blinds where appropriate

$$\mathrm{Enc}_k(\alpha x)$$
$$\mathrm{Enc}_k(\alpha x^2)$$
$$\ldots$$
$$\mathrm{Enc}_k(\alpha x^d)$$

$$\pi = \mathrm{Enc}_k(\mathrm{PCP}(\vec{x}))$$

3. Verifier checks content by simply decrypting

$$\mathrm{Dec}_k\big(\mathrm{Enc}_k(\mathrm{PCP}(\vec{x}))\big) = \begin{cases} 1 & \text{if valid} \\ 0 & \text{if invalid} \end{cases}$$